

Semantic Annotation of Computational Components

Peter Vanderbilt¹ and Piyush Mehrotra²

¹ AMTI*, NASA Ames Research Center, M/S T27A-2, Moffett Field, CA 94035
pv@nas.nasa.gov

² NASA Ames Research Center, MS 258-5, Moffett Field, CA 94035
Piyush.Mehrotra@nasa.gov

Abstract. This paper describes a methodology to specify machine-processable semantic descriptions of computational components to enable them to be shared and reused. A particular focus of this scheme is to enable automatic composition of such components into simple workflows.

1 Introduction

Part of the vision of the Semantic Grid is to enable “an infrastructure where all resources, including services, are adequately described in a form that is machine-processable”[1]. This paper describes a methodology to specify machine-processable semantic descriptions of computational components.

The focus of the CRADLE project is to represent the semantics of workflow components, or so called “tools,” with the ideal goal of enabling automatic generation of semantically correct workflows. A prototype of CRADLE has been implemented including a repository for tool descriptions, a plan (or workflow) generation program and a prototype plan execution system. While the focus of CRADLE is on tools, we think that similar techniques apply to data collections and web/grid services.

In general, a *tool* is a program, component or service that computes one or more outputs from one or more inputs. Some tools require significant computation such as a simulation that computes

* This work is supported by the NASA Advanced Supercomputing Division under Task Order A61812D (ITOP Contract DTTS59-99-D-00437/TO #A61812D) with Advanced Management Technology Incorporated (AMTI).

a flow field around some object or one that computes thermodynamic properties given a flow. However, tools also include format translators, data extraction functions, data analysis and visualization programs.

In order to manage the growing number of these tools and to share them, it is desirable to put tool descriptions in web-accessible, searchable *repositories*. Associated with each tool would be information about what it does, how to run it, who created it, its properties and so on.

The main focus of our research is to find ways to express the *semantics* of a tool in a machine-readable way. Semantic tool descriptions could be used to narrow the set of tools that a user must select from to solve a particular problem. Further, we are interested in *plans*, collections of interconnected tools, and how to use semantic descriptions to ensure the well-formedness of a plan and to determine its semantics. Plans are a kind of *workflow* that have been restricted to DAGs (directed acyclic graphs).

Our goal is to facilitate *plan generation*, which is a process that creates a plan whose execution produces a specified result given specified inputs. Plan generation uses tool descriptions to determine feasible combinations of tools. In order to push the research, we take as the ideal goal the automatic ("hands-off") generation of plans. However, we recognize that, for various reasons, user input may be needed to select among several combinations of tools.

This paper describes CRADLE's *dataspace* approach. A *dataspace* is an abstraction that allows one to provide meaningful, compound names to various properties of real or logical entities. We present how this approach can be used to specify descriptions of computational components and also to chain together these components in semantically consistent ways to yield simplified workflows.

2 Tool and Problem Specifications

Consider the situation where a scientist is computationally investigating the flow characteristics of a class of aerodynamic structures. For a given system of a body and external characteristics, there may be several potential properties that could be calculated. Assume there are tools that take various properties (radius, angle, velocity)

and generate datasets (such as meshes), tools that take datasets to other datasets (such as other meshes and flow simulations), tools that analyze datasets (yielding floats, integers and booleans) and tools that convert between the different dataset formats used by these tools.

The scientist needs a way to compose these tools depending on what is given and what is to be calculated. In order to do this, there must be ways to describe tools and the problems that users want to solve.

Consider a simplified example of a “body” whose properties include its geometry file, volume, mass, (average) density, velocity and momentum. A “problem” specifies which properties will be provided (the “givens”) and which will be calculated (the “goals”). At different times, there will be different sets of givens and goals. Assume that the scientist has at his disposal the following “tools”:

1. Three tools for calculating each of momentum, velocity or mass of a body, given the other two. (There is one tool for each unknown).
2. Three tools for calculating each of average density, mass or volume, given the other two.
3. A tool that takes a body’s geometry file and yields its volume.
4. A tool that calculates the mass and volume of a system of bodies, each with mass and momentum.

So how does one specify these tools and how would the scientist specify what data he wants (given what data he has)?

In a programming language, a tool would be a method or function. In WSDL[5], such a tool would be a web service[4] operation. In these two systems, the machine-readable aspects of a “tool” description are the types of its inputs and outputs. A type system ensures that a composition of tools is consistent at the type level.

While type correctness is required, it is not sufficient for our purposes. For instance, the first three tools mentioned in item #1 above, take two floats and return a float (assuming velocity and momentum are scalars) and are thus indistinguishable from the point of view of the types. Of course, the problem is that the semantics of each tool is not taken into account. Generally, the semantics of a tool is expressed as a comment and/or is implied by the name of

the function and the names of its inputs. It is difficult to effectively reason about information in this form.

Another possible method is to require that the function name indicate the quantity being produced. Unfortunately, there may be tools with more than one output. Item #4, above, is an example of such a tool that generates both mass and momentum.

One can associate one or more output names with each tool. However, there may be more than one tool producing the same output. For instance, there are three tools yielding mass in our example. While the types are different, one could imagine examples where the types are the same. A way to solve this problem is to treat the input names as significant (as well as the output names).

While this approach goes a long way, there are still a few problems. One is that a name alone may not adequately specify a quantity and its representation. For instance, if a velocity-producing tool and a velocity-using tool use different units or formats, the tools will not interoperate. Basically, this is a human issue – the system can compare names but only humans can ensure that the names are used consistently. Thus it is important that each name have an associated natural language description that, together with its type, unambiguously describes the associated quantity and its format.

There is also a problem when two different users unintentionally use the same term with different meanings. While it is possible for a system to detect conflicting definitions, it is better to have a name qualification scheme such that two independent people use different qualifiers for the names they define. Examples of such schemes include UUIDs, URIs, XML QNames and Java class names.

A related problem arises, for example, when one tool uses “width” while another uses “breadth” for the same concept. In this case, the two tools will be deemed incompatible when really they are compatible. What would help is a way to equate two names.

There is also a problem with the “flatness” of names. Consider a simulation of several interacting physical bodies, each with velocity, a mass and momentum. In this case, a simple name, like “velocity,” is ambiguous, since each body has a velocity. One could use names like “b1.velocity” and “b2.velocity” but then a tool taking “velocity” as an input will not apply. What is needed is a notion of compound names, or *paths*. In the system above, the bodies could be named

“b1” and “b2” and their velocities would be named by “b1.velocity” and “b2.velocity” (where the period combines two names). Tools apply as if universally quantified over paths, so the tool described above also can take inputs “b1.velocity” and “b1.mass” and yield “b1.momentum.”

A final problem is illustrated by considering the computation of the area of rectangles and ellipses. Both have “height,” “width” and “area” properties, all floats, but the tools used to compute their areas must be different. The issue is that property names alone do not determine the semantics of the object of which they are a part. Thus there needs to be a way to associate a tool with a class of objects.

3 The Dataspace Model

In this section, we describe an abstract model called the *dataspace model* that addresses the issues of the previous section. Briefly, a dataspace is a tree-like structure with named slots as leaves. A slot can be thought of as a place where a data value can be deposited. Dataspaces have types that imply a vocabulary of slot names and their semantic interdependencies. Each tool is associated, via a relation called “appliesTo,” with one type of dataspace. Logically when a tool runs, it is passed a dataspace of the appropriate type; the tool retrieves its inputs from certain slots and places its outputs in other slots. Roughly, two tools can be composed only if they both have the same “appliesTo” type and the names of the inputs of one are among the names of the outputs of the other. We now discuss the model in more detail.

A dataspace is used to model some real world or logical *entity*, such as a physical body (of various shapes), a surface, a flow field, a particle or a galaxy. An entity can also be a composite thing like a system of several bodies or a flow interacting with the surface of some structure.

A dataspace is made up of a collection of named *slots* each capable of holding one piece of information, like a float, an array of floats or a filename. Each slot is either empty or filled and, when filled, its content denotes one *aspect* of the entity being modeled. An aspect is some parameter, attribute, property or view of an entity.

Example of aspects are a body's mass, velocity or a reference to a file containing its geometry. Different aspects can be used for different representations or units for the same property.

It is possible that an aspect of an entity is itself a composite entity, in which case the aspect is represented by another dataspace, referred to as a *subdataspace*. In general a dataspace forms a tree with named edges and slots at the leaves. For example, a velocity aspect might be a vector modeled by a subdataspace with x, y and z aspects. Similarly a system with two bodies could be modeled as two subdataspaces named "b1" and "b2." In this case, "b1.velocity.x" names a slot that contains the x component of b1's velocity.

Each slot or subdataspace is considered one aspect and is given an *aspect name*. A compound name, like "b1.velocity.x," is called an *aspect path*.

Aspects are typically interdependent and, so, the values of certain slots can be computed from others. For example, a physical body might have aspects mass, geometry, velocity, volume, average density and momentum. Given any two of mass, velocity or momentum, the third can be calculated. Volume, mass and density are in a similar relationship and, presumably, volume can be computed from geometry.

A *dataspace type* denotes a set of dataspaces and is typically associated with some class of entities. A dataspace type defines a vocabulary of aspect names and their associated types and interpretation. The type also denotes a set of constraints on the values of aspects and their interdependencies. These semantic properties are given explicitly by an associated description string or are implied by the names of the type and its aspects. Essentially, the type name becomes a proxy for these human-understood semantics.

Consider the following example definition

```
dataspace Body {  
    aspect URL geometryFile;  
    aspect Float volume;  
    aspect Float mass;  
    aspect Float density;  
    aspect Float velocity;  
    aspect Float momentum;
```

}

While CRADLE actually uses an XML syntax, a more convenient syntax like this is better for explanatory purposes. This definition defines a type named "Body" with six aspects. The first aspect is named "geometryFile" and is of type "URL." The remaining five aspects are of type Float with names "volume," "mass," "density," "velocity" and "momentum." We assume "URL" and "Float" are defined elsewhere. The interdependencies between aspects are implied by the aspect names. A description string could be added to the definition if further explanation was needed.

Each dataspace type definition defines a new, independent type. Even if the aspects are identical, it is assumed that their interdependencies are different, as implied by the name of the type or its description. For instance, there could be two type definitions with identical aspects, "height," "weight" and "area," but having different names, "Rectangle" and "Ellipse." They would denote different types.

The CRADLE type system supports inheritance where inheritance implies an "isa" or subset relation – instances of a derived type are instances of the base type. A derived type has all the aspects of the supertype and can add new aspects, refine existing aspects and add additional constraints (between aspects). For example, Square could be a subtype of Rectangle, adding the constraint that the "height" and "width" aspects are the same. An aspect is *refined* if the derived aspect's type is a subtype of the base aspect's type and if any description-implied constraints of the derived aspect imply the corresponding constraints of the base aspect.

Now that the dataspace model has been described, we turn to CRADLE tool descriptions, which use the dataspace model as a basis for defining their inputs and outputs. Each tool description has an *appliesTo* attribute, a set of input aspect paths and a set of output aspect paths. Consider the following.

```
tool momentum_calc {  
    appliesTo Body;  
    input mass;  
    input velocity;  
    output momentum;
```

...
}

This definition describes a tool that yields the momentum of a body, given its mass and velocity. The ellipsis is to indicate that there may be other attributes for the tool, such as execution information.

The "appliesTo" attribute identifies a dataspace type and specifies that the tool is capable of computing aspect values relative to the associated kind of entity. Thus the "appliesTo" type scopes the tool's inputs and outputs.

The "appliesTo" type also determines the relative semantics of the inputs and outputs. Recall from section 2 the example of two area-computing tools with the same inputs and output, one for rectangles and one for ellipses. In this case, the two tool descriptions would be the same except one would have "appliesTo Rectangle" and the other "appliesTo Ellipse."

4 Plan Generation and Execution

As discussed above, a dataspace type defines a vocabulary of slot names and the semantics of their interdependencies. A tool is specified with respect to some dataspace type and, so, its semantics is determined by the relative semantics of its inputs to outputs.

When a user uses CRADLE, he presents a *problem* which consists of a *problem type*, a set of *givens* and a set of *goals*. The problem type is a dataspace type and each given and goal is an aspect path relative to the problem type. During this process, the CRADLE repository may be used to browse the set of types and their aspects. An example of a problem is as follows.

```
problem {  
    problemType Body;  
    given velocity;  
    given geometryFile;  
    given density;  
    goal momentum;  
}
```

Given a problem, the CRADLE *plan generator* attempts to find a *plan*, which is a directed acyclic graph of *steps*. Each step contains

the name of a tool and the set of *prerequisite* steps that it must follow. The plan also indicates which steps yield one or more of the goals. The plan must be such that each tool's "appliesTo" type is a supertype of (or possibly equal to) the problem type. Each input of each tool must be among the problem's givens or among the outputs of a previous tool. Each of the problem's goals must be among the outputs of some tool.

The full plan generation algorithm is too complex to present here, so we give a quick summary. The algorithm uses backward chaining and works back from the goals. At each point it has a list, *neededAspects*, of aspects (really aspect paths) that need to be computed and a list, *producedAspects*, of aspects that are given or computed by some tool. Iteratively, it selects a needed aspect, *subgoal*, and finds a tool, *tool*, whose "appliesTo" is a supertype of (or equal to) the problem type and whose outputs contain *subgoal*. If there is no such tool, it backtracks if possible. If there is more than one tool, it tries each in turn. To handle the "flatness" problem mentioned in section 2, the algorithm also considers applying a tool to certain subdataspace of the original problem, in addition to applying it at the root. The outputs of *tool* are added to *producedAspects* and its inputs are added to *neededAspects*. Also a step is allocated and added to the plan. The iteration terminates when all *neededAspects* are in *producedAspects*.

Plan execution is the process by which a plan is executed. It follows the usual rules for executing a DAG. The dataspace model is used to link outputs of one tool to the inputs of the next. As mentioned earlier, the dataspace concept is logical and it is not necessarily the case that any real data structure directly implements the dataspace, although some implementations may. The purpose of the dataspace concept is to provide a conceptual model interrelating types, tool descriptions, problem specification, plan generation and plan executions.

As examples, one implementation may directly implement the dataspace as a centralized hash table from which the tools extract their inputs and into which they place their outputs. Another implementation may instantiate a software component for each step and use the fully qualified input and output names to hook together ports. A third implementation might generate a script using a "man-

gled" form of the aspect paths to name files or script variables that carry data produced by one step to later ones.

5 Status and Future Work

A CRADLE prototype has been implemented using a client-server model with a protocol similar to web services. The server is in Java and accesses a MySQL database containing tables for tool and dataspace type descriptions. The type descriptions can be used by tool specifiers and by users posing "problems" to CRADLE. Type descriptions are also used during plan generation to reason about inheritance and the types of subdataspaces. Tool descriptions are used during plan generation and may also be used during plan execution to obtain execution-related information.

For future research, we will look at applying a similar methodology to data collections and the tools that operate on them. As RDF[3] is a popular standard for expressing and sharing machine processable information on the web, we will investigate using RDF/XML[2] in the client-server protocol. Also, we plan to look at various extensions to the dataspace model, including arrays, parameterized aspects (similar to methods) and parameterized types. Adding machine-processable constraint expressions to dataspace types is another potential avenue of investigation.

Acknowledgments: We gratefully acknowledge the contributions of Ken Gee and Karen McCann in the early discussions regarding the direction and design of the project.

References

1. Global Grid Forum, Semantic Grid Working Group: The Semantic Grid Vision. <http://www.semanticgrid.org/vision>
2. W3C: RDF/XML Syntax Specification (Revised). <http://www.w3.org/TR/rdf-syntax-grammar/>
3. W3C: Resource Description Framework (RDF): Concepts and Abstract. Syntax. <http://www.w3.org/TR/rdf-concepts/>
4. W3C: Web Services Architecture. <http://www.w3.org/TR/ws-arch/>
5. W3C: Web Services Description Language (WSDL). <http://www.w3.org/TR/wsdl20/>